Ray – A Secure Micro Kernel Architecture

Stefan Nürnberger, Thomas Feller, Sorin A. Huss
CASED - Center for Advanced Security Research Darmstadt, Darmstadt, Germany
{stefan.nuernberger | thomas.feller | sorin.huss } @ cased.de

***Abstract*—In this paper we present a secure micro kernel architecture (called Ray) that was designed from scratch with security goals in mind. It features some traditional security aspects like process isolation, advanced non-standard security aspects like padded non-readable memory boundaries and new contributions like memory gifts and behaviour deviation detection. This theoretical design has been implemented as a proof of concept for x86 based processors including a small set of essential drivers and user land applications in order to verify and test the claims made herein.**

***Index Terms*—operating system, secure micro kernel, harvard architecture, message passing, behaviour analysis**

I. Introduction

As the operating system is the fundamental base and interface to the hardware, it does not only implicitly and explicitly define security boundaries, but it is also an important link in the chain of a secure architecture. If an application poses a security threat due to a vulnerability, the system may be vulnerable when this particular application is running. In contrast to that, the operating system is always running and should be designed to prevent malicious and accidental action that might pose a security threat by design.

The micro kernel architecture and implementation described in this paper was designed from scratch in order to be able to see the implications, when completely focusing on security aspects and intentionally neglecting compatibility influences. Furthermore, a small trusted code base enables inspection of integral parts separate from each other under almost laboratory conditions. This enabled us to re-think traditional operating system design and implementation in order to guarantee mission critical scenarios the utmost trust in the underlying OS.

The main goals of our approach are

- Common programming mistakes shall be easily discoverable – even when using non-managed languages like C.
- Security vulnerabilities (especially code injection by several means) shall be impossible.
- Dependabilities between applications shall be minimized (as they cannot be completely avoided).

We have implemented a proof of concept version for the Intel x86 architecture to test the theoretical claims made herein.

II. Related Work

Most previous work made additions to already existing operating systems or kernels, like IBM's Integrity Measurement Architecture – IMA[1], which adds Trusted Boot capabilities to the Linux kernel. Flume [2], a user mode reference monitor for Linux, allows Distributed Information Flow Control (DIFC) implemented in user space. Complete kernels have been implemented too, mostly focusing on one or a few security aspects, like HiStar [3], which also implements Information Flow Control (IFC) to achieve a small trusted code base and even an entirely untrusted login process. Only a few kernels exist that try to combine best practices of secure kernel design with new technologies to get an overall picture of how an operating system might look like, when security aspects have major priority. One of the most interesting projects is Microsoft Research's Singularity OS ([4], [5]). Our paper focuses on a similar complete overview and design of a secure, trusted and dependable operating system.

III. General Kernel Design

The kernel was almost entirely written in C and C++ in order to benefit from higher level languages and especially their constructs like C++ exceptions. In order to make C++ –including run time type information (RTTI) and exceptions– work, the standard C library (using newlib[1]) and the C++ support library (libsupc++ from GCC) was ported to be able to run in kernel mode. This emerged to be a tricky task, as operators like *new* or *delete*, exception handling as well as stack unwinding heavily rely on operating system functionality, which is a chicken-egg-problem. The current implementation is partly based on the work presented in [6].

Due to its object oriented nature, C++ is used when the kernel benefits from it (like abstract data types to manage hash maps), while plain C is used to write the C++ support code as well as basic functionality. The hardware abstraction layer (HAL) of the kernel has minor parts written in Assembly when necessary. The assembler instructions make up $\approx 1.5\%$ of the overall instruction count of the kernel image, but only $\approx 0.3\%$ lines of code. Table I shows an overview of the different parts and their amount of lines of code (LOC) that compose the kernel. The kernel compiles to $\approx$ 130 KiB of x86 instructions without optimization ($\approx$ 100 KiB when optimized for size). Of course, it features traditional security features like separate processes and threads and isolation of processes using hardware virtual memory management.

A. Boot Process

The kernel image complies to the multiboot [7] specification. We use a standard, unmodified Grub or Syslinux[2] to boot

[1] http://sourceware.org/newlib
[2] http://www.gnu.org/software/grub and http://syslinux.zytor.com

TABLE I
SOURCE CODE SIZE

Part	Source Code Lines	Fraction
Memory Mgmt.	477	5.74%
Scheduler	366	4.40%
Abstract Data Types	315	3.79%
IPC	286	3.44%
Bootstrap	262	3.15%
Synchr. Primitives	219	2.64%
C++ Support	201	2.42%
Syscalls	100	1.20%
HAL	785	9.45%
Headers	2868	34.51%
Miscellaneous	2431	29.25%
Sum	8310	100%

up the kernel from either hard disk, USB flash media, CD-ROM or over the network via PXE. It is designed to be one piece of code that does not need to be boot-strapped. This is addressed by abusing the Global Descriptor Table (GDT) in order to shift the kernel to 3GB right from the beginning. This is necessary as the kernel image is statically linked to work at 3GB, which would apparently be impossible if the user did not offer this amount of memory. Once virtual memory is initialized, the GDT offset is removed and the MMU is responsible for mapping the kernel at 3GB, regardless of the amount of physical RAM installed. Traditionally this is done by chain-loading at least two pieces of differently linked kernel parts. Our approach eases maintainability and makes the kernel less prone to boot loader errors.

B. Hardware Access

Due to its nature as a micro kernel, the drivers run as ordinary user mode processes with special privileges to register deferred procedure calls for interrupt handlers (as we do not want them to take place in kernel mode) and restricted I/O. That means, that I/O access has to be granted by the kernel on a per-port basis. The kernel further ensures that one I/O port is only accessed by exactly one process. Therefore, claims about concurrent access can be made, as the access to a common I/O port (like the PCI bus) has to be tunneled through a separate PCI bus process. As using a kernel API to access each port would be comparatively slow, our implementation uses the x86 feature of the *IO-Bitmap* to restrict access to all ports except the ones allowed and registered for.

C. Behaviour Deviation Detection

In order to also assure known behaviour, our approach is to dynamically recompile each process before its gets started. The kernel disassembles the instructions of a new process and injects extra monitoring algorithms. They verify return points from functions, to check that they fall back to exactly the point they came from (see return to libc attacks in [8] and why Address Space Layout Randomization – as in use by Linux or Windows – is not a good choice [9]). They can also be used to assure that a program executes and behaves according to its learning phase. The kernel inserts checkpoints (using the CPU's debug registers) to every conditional branch and function call / return pair. When triggered, these checkpoints assure known behaviour with respect to an earlier learning phase. The algorithm can abstract from loops, recursion and asynchronous behaviour like threads. Only the paths and their coverage is checked. During the linear control flow between each conditional branch, additionally inserted data monitoring algorithms take care of detecting buffer overflows (see section IV). By this means, it is possible detect deviation from intended behaviour due to erroneous programming or security vulnerabilities.

D. Performance

The following table gives an overview of cost of basic kernel operations. The tests are identical to the tests in [10, p. 11] where the Singularity, Windows and Linux values have been taken from. *ABI call* refers to the most simple kernel ABI call which returns a readily available data structure. *2 thread sync* benchmarks the complexity of a wait/notify synchronization primitive of two threads in the same process. *IPC message* is the time it takes a process from sending a 1-byte-message till it can be read in another process. *Create process* is the time from the call of *ProcessLoad()* till the first instruction in the newly created process. All tests have been conducted 1000 times and averaged. The benchmarks were performed on an AMD Athlon64 3000+ (1.8 GHz) CPU with 1GB of RAM. Please note, that the worse performance of IPC messages can be explained by the benefits over traditional IPC (see V).

TABLE II
BASIC OPERATION PERFORMANCE

Operation	CPU Cycles			
	Ray	Singularity	Linux	Windows
ABI call	251	91	437	627
2 thread sync	685	346	906	753
IPC Message	11,683	808	5,797	6,344
Create Process	120,661	352,873	719,447	5,375,735

IV. MEMORY

The Ray memory management deviates from traditional management as the memory is divided in two independently managed areas. Despite the fact the implementation was developed on an x86 platform, which features a Von-Neumann architecture, this model enforces a Harvard architecture by combining virtual memory and memory segmentation. Kernel and user mode code is only allowed to run within a certain segment. The data segment overlaps the complete code segment. This enables code that tries to read its own instructions. It does not pose a security risk, as the code segment is write-protected. However, instructions synthesized or copied from the code segment cannot be executed, as execution of them would have to take place outside of the code segment.

As a means to prevent inadvertent access to variables not intended to use, the kernels marks its whole data segment as write-protected. As all the kernel functions – especially system calls – only modify data structures (create process, allocate data, allow I/O access, . . .) in a fairly small fraction of their overall work, it was a viable decision to surround access to

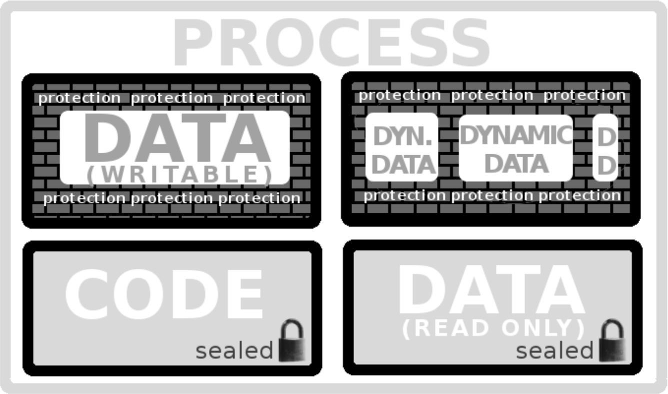
PROCESS
protection protection protection
DATA
(WRITABLE)
protection protection protection
protection protection protection
DYN. DATA
DYNAMIC DATA
D D
protection protection protection
CODE
sealed
DATA
(READ ONLY)
sealed

Fig. 1. The standard process memory layout: Sealed data cannot be extended or changed, writable data cannot be extended and dynamic data can be written to and extended but its boundaries are protected.

data structures with an unlock/lock pair. These unlocks/locks only update one entry each in the Translation Look-aside Buffer (TLB) using directly manipulating CPU instructions[3].

A. Shared Memory

Ray user mode processes have the restriction that they are not allowed to share memory between them. On the one hand this eliminates a common programming pattern – shared memory – but on the other hand improves static code analysis as pointers are guaranteed to be only modified by the owner and they are restricted to point to addresses inside the program only. The loss of message exchange capabilities is explained in section V.

B. Memory Protection

The Ray memory management has two more techniques to confine common programming techniques related to memory and arrays. When not using a higher level programming language which lacks a runtime virtual machine (like Java or .NET CLR), access beyond the boundaries of an array may occur unnoticed. This may not only lead to corrupt data as other data may have been overwritten, but it exhibits unwanted behaviour. To circumvent this in a majority of cases, the Ray kernel aligns a requested memory block so that its last byte is at the end of a virtual memory page. Two more adjacently surrounding pages are introduced at either side of the allocated memory block. These pages are marked as read- and write-protected. So **any access** (either in kernel mode or user mode) will trigger a page fault and will therefore be noticed – even if the index in the array is only off by one (see figure 1).

This method cannot cope with arrays that are statically allocated at compile time and reside in the *data* or *bss* segment of the executable. However, we introduce an algorithm that copes with this kind of problem. The dynamic recompilation introduced in III also assures that no arrays have been overwritten by emitting and compiling two different – but semantically identical – programs, that are being run side-by-side in lockstep (similar to [11]). One version has no modifications and works as a reference, while the second version is changed, so that all pointers to the *data* section are commulatively moved to introduce a gap of at least one byte next to each other. Of course, the actual data in the *data* sections is splayed accordingly. During executing all values read from the data section are efficiently accumulated (i.e. using an XOR operation) in an unused register in both process versions. Thus, exceeding array boundaries leads to different values of data bytes following the overwritten value. Thus, on the next read access different values will be read and therefore be accumulated. These accumulated values are compared at each conditional branch and the process can fire an exception in case they differ.

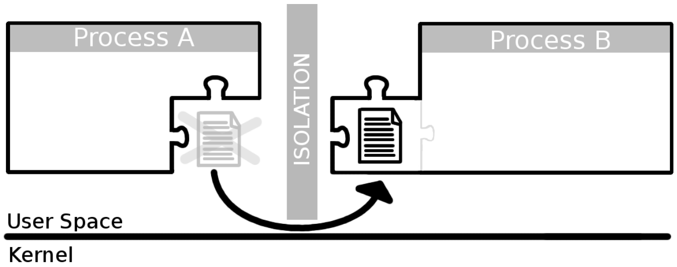

Fig. 2. IPC Memory Gift: Donating memory from process A to process B. The sender loses virtual memory that the receiver gains.

V. Inter Process Communication

In contrast to commercial operating systems, the Ray micro kernel features a different approach for Inter Process Communication (IPC). While Unix/Linux traditionally use signals and pipes to signal respectively receive notifications, Windows uses a method usually referred to as the *Message Pump* to receive signals and data in user mode applications. We think that both methods incorporate drawbacks concerning security or general correctness. In order to signal a process the Unix/Linux kernel simply pretends a function call in the respective process to a previously registered signal-handler. This involves an issue of general applicability as the signal handler can interrupt at virtually any point, thus potentially interrupting non-reentrant code. The Windows operating system copes with that fact by using the aforementioned *Message Pump* which has to be checked and emptied in an infinite loop. On the one hand this approach avoids the interruption of non-reentrant code, but on the other hand urgent events can apparently not be signaled immediately when an old event is still being processed.

To circumvent the two above mentioned drawbacks, the Ray micro kernel features IPC based on signals that are started as separate threads. So functions incorporating critical sections can still protect them, even though they get interrupted in the middle of execution. Additionally due to the nature of a thread, signals can be prioritized implicitly by the scheduler because they have an associated priority which may interrupt a currently running lower priority thread. Nevertheless threads of equal priority level are guaranteed to arrive in order of sending. Ray features IPC mechanisms that are derived from message passing of traditional micro kernels and are in some degree

[3] Using the *Invalidate Page* instruction *invlpg*

similar to the concept of channels introduced in Singularity [12]. IPC messages have exactly one sender and one receiver and must agree on an exchange format prior to using the IPC channel (see the principle of *Contracts* in [12]). The message format they both agree on includes the actual message type and a maximum capacity/length the sender is willing to expect. In case of integer or floating point values, they are copied in registers. Otherwise the messages are completely donated to the receiver's process. Those IPC messages are therefore called **Memory Gifts** as the messages leave the sender's process and arrive unchanged without the need to copy them (see figure 2). This is done by simply changing the ownership of the particular memory block from sender to receiver. To enable this technique, memory that can be donated between processes is marked for ownership exchange by the kernel. Therefore it has to be allocated using dedicated functions alongside the usual *malloc/free* pair, thus, exchanging usual local variables (stored in the .data, .bss segment or on the stack) and normal memory allocated via *malloc* is prohibited in order to avoid inadvertent donation of memory. After a message has been sent, the sender's process has a dangling pointer as the kernel immediately flushes the TLB cache so that the just donated memory cannot be accessed accidentally. A received message is still marked for exchange and can of course be forwarded to yet another process.

This method has several advantages.

1) As a fire-and-forget type message, no implicit expectations exist about the handling of messages and their protection using mutual exclusion.
2) After the unexpected death of a process, shared memory would have been left in an undefined state, leaving no other option than to terminate all the processes involved in the sharing of memory. If either side of two processes exchanging messages dies, no other process has to be terminated.
3) The use of a variable/memory after it has already been sent can be easily trapped by hardware mechanisms as they generate a page fault due to dangling pointers.
4) Moving large data structures through several abstraction layers is rather efficient despite the micro kernel design, as the memory blocks are simply passed through each layer (e.g. IDE drive → DMA → disk driver → file system driver → application).

VI. Conclusion

Our approach features means to avoid code injection by prohibiting the execution of stack or data in general. Its IPC mechanisms enable stronger claims when using static analysis due to a lack of potentially harmful shared memory. The message passing using threads protects non-reentrant code and critical sections. Driver's I/O access can be allowed in a fine-grained manner and the kernel is able to detect array-out-of-bound programming flaws, at least when using dynamically allocated data. The proof of concept was written in a high level programming language (C++) with all its benefits like object orientation and exception handling, in order to further reduce programming flaws.

The kernel represents an architecture that ensures secure execution from tip to toe and provides a solid basis for further research.

VII. Future Work

Our current and future work concentrates on completely eliminating code injection (including return to libc attacks). As a basis, already existing means like Secure Boot (see TCG[4]) will be implemented as described by [13], to ensure the kernel's integrity. A second step will be to only allow drivers or even applications to run that have been digitally signed, so that their integrity can be proven. These approaches however do only guarantee that the integrity of a previously known-good state has not been compromised.

The dynamic recompilation and monitoring according to a previous learning phase could also be widened in order to intentionally restrict access to certain features of a program of which modification of the source code is (no longer) possible (e.g. commercial applications).

References

[1] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 223–238.

[2] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard OS abstractions," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, p. 334, 2007.

[3] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proc. of the 7th OSDI*, pp. 263–278.

[4] G. Hunt and J. Larus, "Singularity Technical Report 1: Singularity Design Motivation," Technical report, Redmond, WA, Tech. Rep., 2004.

[5] G. Hunt and J. Larus, "Singularity: rethinking the software stack," vol. 41, no. 2. ACM, 2007, p. 49.

[6] H. Isak Gylfason and G. Hjalmtysson, "C++ Exceptions & the Linux Kernel-C++ kernel-level runtime support for Linux lets you use the full power of C++ in kernel-space programming," *Dr Dobb's Journal-Software Tools for the Professional Programmer*, pp. 50–53, 2005.

[7] Y. Okuji, B. Ford, E. Boleyn, and K. Ishiguro, "The multiboot specification," 2002.

[8] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, p. 561.

[9] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2004, pp. 298–307.

[10] G. Hunt, C. Hawblitzel, O. Hodson, J. Larus, B. Steensgaard, and T. Wobber, *Sealing OS Processes to Improve Dependability and Safety*. Proceedings of the European Conference on Computer Systems (EuroSys), Association for Computing Machinery, Inc., Lisbon, Portugal, 2007.

[11] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proceedings of the fourth ACM european conference on Computer systems*. ACM New York, NY, USA, 2009, pp. 33–46.

[12] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi, *Language Support for Fast and Reliable Message-based Communication in Singularity OS*. Volume 40, Issue 4 Proceedings of the 2006 EuroSys conference, 2006.

[13] W. Arbaugh, D. Farber, A. Keromytis, and J. Smith, "Secure and reliable bootstrap architecture," Feb. 6 2001, uS Patent 6,185,678.

[14] S. Engle and M. Bishop, *A Model for Vulnerability Analysis and Classification*. IEEE Conference on Computational Science and Engineering, 2008.

[4] Trusted Computing Group – http://www.trustedcomputinggroup.org